Journal of Physics: Conference Series **396** (2012) 052071 doi:10.1088/1742-6596/396/5/052071

# Cling – The New Interactive Interpreter for ROOT 6

**V Vasilev[1], Ph Canal[2], A Naumann[1], P Russo[2]**
1 CERN, PH-SFT, Geneva, Switzerland
[2] FermiLab, P.O. Box 500 Batavia, IL, US

E-mail: vasil.georgiev.vasilev@cern.ch

**Abstract.** Cling is an interactive C++ interpreter, built on top of Clang and LLVM compiler infrastructure. Like its predecessor Cint, Cling realizes the *read-print-evaluate-loop* concept, in order to leverage rapid application development. Implemented as a small extension to LLVM and Clang, the interpreter reuses their strengths such as the praised concise and expressive compiler diagnostics. We show how to match the interpreter concept to the compiler library and generalize common set of requirements for building up an interactive interpreter. We reason the design and implementation decisions as solution to the challenge of implementing interpreter behaviour as an extension of the compiler library. We present the new features, e.g. how C++11 will come to Cling and how Cint-specific extensions are being adopted. We clarify the state of integration in the ROOT framework and the induced change set. We explain how ROOT dictionaries are simplified due to the new interpreter.

## 1. Introduction

Rapid application development (RAD)[1] becomes essential in both scientific and industrial world. The ability to develop quickly software prototypes and proof-of-concept applications is the shining beacon for success in many projects. Prototyping is an effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design and implementation. However, it depends on the time consumed by edit-run cycles during development. Compiled languages are even a bigger burden, since much time is spent in compilation and linking.

In general, compilation and linking slowdown could be avoided by using an interpreter but the inconvenience of running every time for just a test stays. Even though whether a language to be interpreted or compiled is a property of the implementation, in practice there are not many programming languages to offer both – a compiler and an interpreter translating them into machine code.

Written mainly in C++, ROOT[3] is a huge framework consisting of libraries for storing, analysing, processing and visualizing data. According to the latest estimates ROOT is used to store more than 120 petabytes of data, which makes it a central pillar in high-energy physics. ROOT is designed to operate with great amounts of data and it is widely used outside the HEP field.

Most of its users are non-experts and thus it has to offer them a way to develop their applications quickly and effortlessly, too. In order accomplish that, ROOT leverages the software technique rapid application development, or used in short – rapid prototyping. A key component responsible for realizing such exploratory programming style in ROOT is its interactive C++ interpreter – Cint[4].

[1] To whom any correspondence should be addressed.

Cint was designed and developed to be a C interpreter, but later was patched with C++ support as well. Besides not being modular and flexible, one of its major shortages is it conforms neither C nor C++ standard. Consequently it is not going to conform the new standard – C++11.

ROOT relies broadly on its C++ interpreter. While Cint is thought as the ROOT's interactive prompt, its most important use is to deliver description of the data that needs to be serialized and deserialized by ROOT's input/output (IO) subsystem.

This paper proposes a new state-of-art C++ interpreter, Cling – incorporating more than 15 years experience in development and maintenance of Cint and overcoming Cint's limitations. In its design and development the new C++ interpreter considers one of its major use cases: ROOT. However, Cling can be used as standalone C++ interpreter in a context distinct from ROOT and the field of high-energy physics.

The paper is divided into 6 sections: Section 2, Background and related work, describes the tools and technologies that simplify construction of interactive interpreters. It explains in brief the interactivity's cornerstone – the read-evaluate-print-loop (REPL) concept. It gives a glimpse of similar tools; Section 3, Cling overview, shows the set of technologies the new interpreter was built on; Section 4, Cling in a nutshell, is logically split into two subsections. The first part generalizes the common features and properties of an interactive interpreter. The second part illustrates how they were implemented in Cling. Section 5, Integration in ROOT, points out a set of changes, caused by Cling's adoption in the framework; Section 6, Conclusion and future plans, presents a summary and discusses some interesting developments that are possible in the context of the new interactive interpreter.

## 2. Background and related work

Modern compiler platforms and architectures (such as LLVM[2]) allow implementation of hybrid compilers[5](pp 1-3). The source language is compiled into machine-independent intermediary representation (*LLVM bitcode*), which at execution time is again compiled into machine code. The compilation at runtime is done by Just-In-Time (JIT) compiler[5] (pp 507-509), blurring the boundary between compilers and interpreters.

Emulating interpreter behavior becomes simpler by using the present-day compiler platforms and architectures. Its cornerstones are the hybrid compilers and incremental compilation. The hybrid compiler must translate the code line-by-line into memory and translating each line it must ask the JIT to compile and run it. That set of functionalities enables an interpreter to be built on top of a compiler platform as a tiny set of supplementary libraries.

Conversely, building an interpreter and making it user-friendly and usable are different stories. Often the terms user-friendliness and usability boil down to improving the tool-user interactive communication. Unfortunately these properties are hard to quantify. There are a few interactive interpreters, which offer a shell-like command prompt. In order to become useful and user-friendly they adopt the REPL concept[6]. Among them, well-known are CH[8], Cint (C/C++) and CSharpRepl[7] (C#). CH is a cross platform C and C++ interpreter with built in plotting capabilities. Its limitations are similar to Cint's – it doesn't support fully C++. Another example is CSharpRepl, which relies extensively on Mono implementation of the Common Language Infrastructure (CLI)[9] and Mono's hybrid C# compiler. It proves that such interactive interpreter could be built on a different set of technologies using the same methodology.

Usually REPL interpreter extends the programming language by making possible to run *expressions* at global level, which is forbidden by most language specifications. Another common extension is implementation of special commands, which provide information about the current state of the environment. Aliases are different dimension of commands. Aliases are shorter replacement for much longer commands for user convenience.

A user-friendly and interactive (REPL) interpreter could accelerate the development speed and quality. It also helps to deal with the complexity of both framework and programming language. A REPL interpreter becomes an essential ingredient in RAD, allowing even non-expert users to learn quickly and rapidly materialize their ideas. In the context of ROOT this approach was adopted and

proven to work in practice. ROOT provides a command prompt C++ interpreter – Cint, which allows users to explore the framework and helps them even to learn the programming language quickly. A user could start ROOT, prototype an idea and see the results right away. If the developed algorithm is satisfying, the prototype could be compiled for improving the performance and compliance to the C++ standard.

## 3. Cling overview

Cling is a full-blown C++ interpreter considered as replacement for the veteran – Cint. The new C++ interpreter heavily relies on LLVM[2] and Clang[10] compiler platform. From LLVM's and Clang's perspective, however, Cling is not an interpreter but just a tool built on top.

LLVM is an open source, BSD-licensed software delivering reusable compiler infrastructure. It is an umbrella project that hosts and develops a set of close-knit, low-level toolchain components (such as assemblers, compilers and debuggers), which are compatible with existing tools. LLVM provides some unique abilities and is known for some of its widely used tools such as Clang and LLDB[11]. Significant advantage of LLVM, which sets it apart from other compilers, is its internal architecture. Modularity and reusability have always been an important property of software developments, but LLVM brings it to the next level. The cooperation between the libraries remains loose-coupled but without any efficiency trade-off.

Clang is a C/C++/Objective-C hybrid compiler that offers many benefits over the other existing compilers. Key advantages are fast compilation, low memory consumption, expressive diagnostics and GCC compatibility. Clang follows LLVM's reusable library principles in its design and development, allowing various tools to be built on top. For example there are many tools for refactoring, static analysis and code transformation. Users see Clang as a compiler, toolmakers – as a platform. It implements the different compilation phases as separate steps, encapsulated in libraries with stable *application programming interfaces* (API). A tool could reuse only necessary libraries and compilation phases, free of redundancies.

The compiler platform supports all pieces for implementing successfully a C++ REPL interpreter – incremental compilation, JIT, stable API and reusable compilation phases. LLVM and Clang are the perfect environment in which Cling can live and grow. The umbrella project offers serious and flexible continuous integration suite at all stages of translation. The suite can be easily inherited, extended and reused, which is of importance for Cling's health.

## 4. Cling in a nutshell

In the past the REPL concept was brought only to the interpreters, because it was much easier to implement rather than in a compiler. The reason is in the incremental translation and execution, done by definition while interpreting.

Many modern compilers can do incremental translation, which allows the REPL concept to be implemented on top of compiler as well. Since Cling is a REPL, build on top of a compiler it can be called both interactive compiler and interactive interpreter. Those terms are used interchangeably in the paper.

In the section we will describe how to build an interactive compiler such as Cling. We explain how to teach a compiler to be interactive, i.e. behave as an interactive interpreter.

### *4.1. Interactivity*

In order to make the future interpreter interactive and useful we should extend the source programming language. An *interactive language extension* is a place where the language might be enriched to improve the user experience.

From our practice with Cint and considering the related projects in the field the requirements for interactive language extensions converge into a common set. The language of an interactive interpreter should include:

- Error recovery – it has to “survive” the errors made by the user without restarting or having to redo everything from the beginning.
- Expression evaluation – in most programming language expressions and in general statements are only allowed in declarations. For example, variable declaration that contains an expression as an initializer or function declaration, with body containing statements. Allowing the users to type an expressions and statements at the prompt is a huge convenience.
- Streaming out the execution results – another huge ease for the users is to be able to see the results of the previously run command or statement as an instant response from the system.
- Runtime dynamism – a hard to define feature, whose main objective is to improve the user-interpreter interaction with runtime-evaluated entities (dynamic entities).

Implementation of the defined properties advances the user-interpreter experience. It has excellent results even on static languages such as C and C++. Cling targets implementing all these properties in order to be as user-friendly as possible.

### *4.2. Implementation*
In this section we discuss in details the implementation of the properties in section 4.1. They are realized in Cling, LLVM and Clang in C++.

#### *4.2.1. Error recovery*
Shell prompt must work even if errors are found. Cling must inform the user about the error and continue working seamlessly – extremely challenging to achieve in a compiler library. This should not be confused with the primitive error recovery, available in compiler’s parser. The parser error recovery is mainly to filter out false positive diagnostics. The underlying compiler does not know how to recover fully from an error. Usual compiler action flow on finding an error is to issue the proper diagnostic and to try to find other unrelated problems. It does it, knowing that next time it starts it will start from the very beginning of the compiled program.

Cling is unique in that respect because it has to continue to consume new user input as if it has not seen an error ever. One has to consider that a user transaction may contain both – correct and incorrect language constructs. The underlying compiler library will swallow the correct and issue diagnostic for the incorrect ones. This contradicts with the intuitive semantics (**Figure 1**). Natural expectation is the entire input line should be reverted even if it contains correct constructs. This is a reason to call the incoming user input a *user transaction*.

```
[cling]$ int i = 0; error_here<>;
[cling]$ int i = 0; int error_fixed;
```

**Figure 1**. Error recovery example.

If we wanted to preserve the expected semantics we had to revert all the changes the correct constructs induced in the underlying compiler’s internal data structures. One of the most important compiler data structure is its *abstract-syntax-tree* (AST). (The AST is a graph representation of the source code. Interestingly enough the usual meaning of the term AST is very different from its definition[5] (pp 69-70). Usually AST means concrete semantic tree.)

However, this is not enough, because in principle code was emitted for every correct construct. However, Cling implements user transaction caching mechanism to make sure that every construct in the input was correct and then send the cache queue for code generation. If there was an incorrect construct, Cling discards the cache and repairs the intermediary internal compiler data structures (AST, redeclaration chains, identifier resolver chains, etc.).

#### *4.2.2. Expression evaluation*
C++ programming language allows only declarations at the global namespace[12] (pp 161-164). C++ language must be extended in order Cling to be able to evaluate *expressions* and *statements*. The safest

and easiest way is to turn the ill-formed code into its well-defined analogue. Pragmatic manner to do it is by transforming the statement into declaration at textual level (**Figure 2**). Cling preprocesses the input, transforms it into well-defined code and passes it to the underlying compiler library to produce executable code.

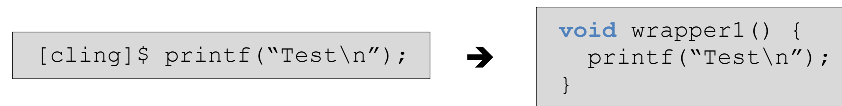



**Figure 2.** Simple source-to-source transformation of ill-formed code.

It is clear the general case is lot more difficult (**Figure 3**). In a more usual case we might have declarations (for example variable declarations). If the declarations are wrapped, their visibility changes and the transformation becomes semantically incorrect. The side effect would be the declarations wouldn't be visible by the incoming user transactions.

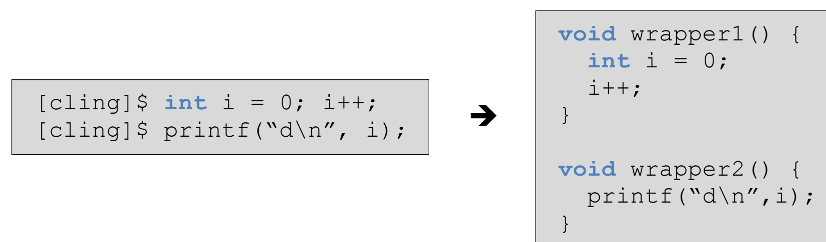



**Figure 3.** General source-to-source transformation of ill-formed C++ code.

The next step that Cling undertakes to preserve the expected semantics is to let the underlying compiler parse the transformed source code without producing any executable code. Then Cling finds the produced code representation corresponding to the wrappers and starts looking for declarations. If Cling finds a declaration inside a wrapper, it is promoted one level up – onto the global namespace and its scope turns into global (**Figure 4**).

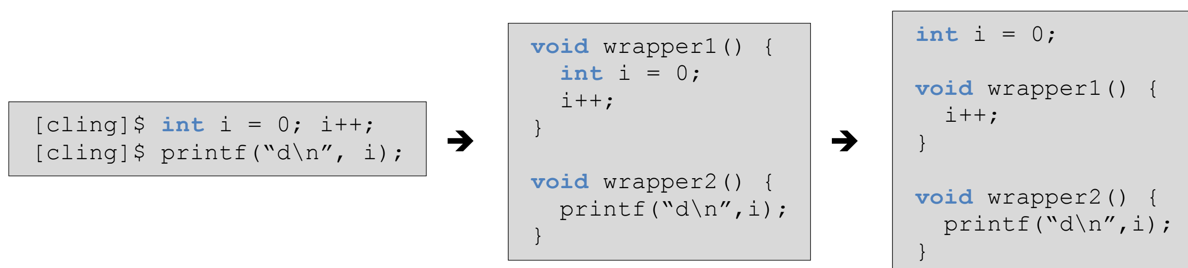



**Figure 4.** Additional syntax tree (AST) transformation preserving the semantics of ill-formed C++ code.

The described steps are enough for most of the cases. However it cannot handle all possible declarations types (defining namespaces, for example). In that situation a simple lightweight lexical analysis is performed. Thus, for instance, this helps moving a namespace on the global scope.

*4.2.3. Streaming execution results*

Typically a user writes a statement for evaluation and expects the statement to be executed and the result to be displayed immediately. Having a switch signaling whether Cling will show the execution result is a central requirement. Thus it was implemented as an interactive language extension, which adopts Cint's way of signaling. The omitted semicolon (;) in the end of a statement asks for the execution result to be printed out.

Internally Cling turns the statement into valid C++ code and marks it as "special". Later when it was parsed and the AST was built, Cling attaches additional complex template machinery, called *expression printer*, to the special statement. The expression printer has to be user-extendable, allowing printing user-defined types. The attached complex machinery provides an interface, which users can implement to customize the printout. The interface contains all necessary information (provided by the AST) about the printed type.

*4.2.4. Dynamic scopes and late binding*

A user interacts with what he imagines to be an interpreter and he expects to be able to load external objects at runtime. Those objects are often called *dynamic objects*. In principle the dynamic objects should be treated and handled as a static object. In terms of C++ and the underlying compiler, complex machinery is needed. This essentially boils down to introducing dynamic scopes[5] (pp 31-33) in C++.

Cling introduces dynamic scopes to C++ to enhance the scripting experience of the language. The feature is useful for frameworks, which embed Cling and provide additional external objects at runtime. A common scenario in ROOT is opening a serialized C++ object and accessing or calling its members (**Figure 5** on the left). The semantics of the example is on successful open, the objects in the file should be added to the static scope. The underlying compiler cannot possibly know what the files contain until the execution at runtime. Consequently it will issue an error that "hist" is unknown name – an undesirable outcome. The effect that we look for has to "enrich" the *current* visibility scope by "injecting" the file contents. When the opened file goes out of scope the injected contents must no longer be accessible. Since "hist" becomes known late – at runtime, this semantics is considered as late binding. Expressions and statements managing dynamic objects are called *dynamic expressions* and *dynamic statements*.

The implementation of the expected semantics has two main actions at two different stages. Some details have been intentionally omitted in the example for clarity.

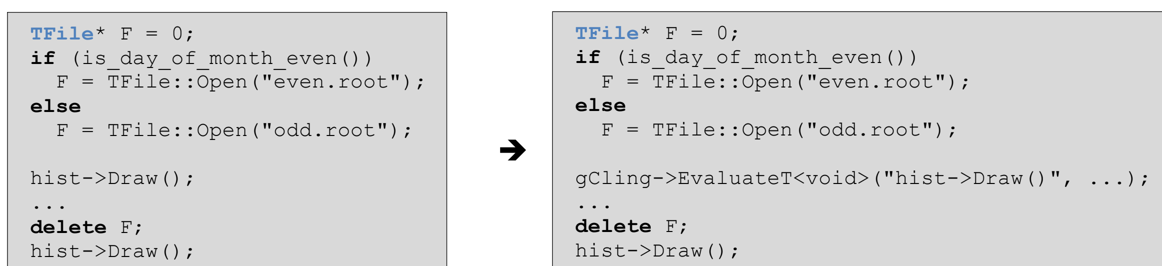


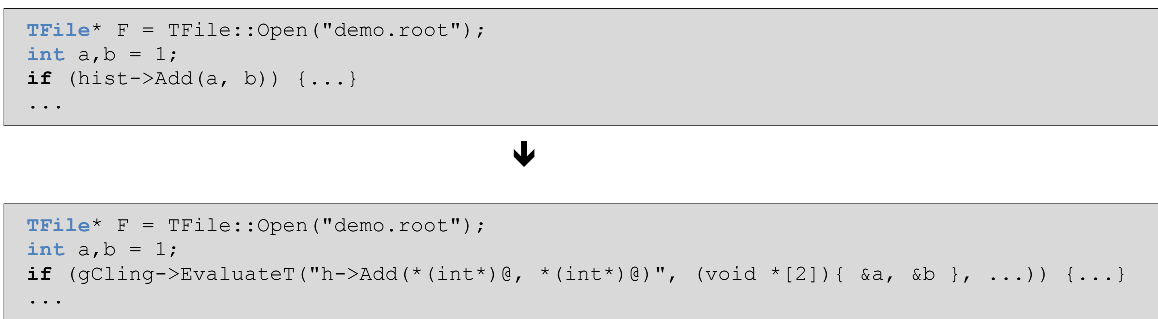

**Figure 5.** AST transformation of ill-formed C++ code preparing it for partial recompilation at runtime.

In brief the first action is teaching the compiler not to issue any diagnostics when it encounters such scenario but still produce a valid AST. It must mark the unknown AST nodes as "to be fixed later" by someone who can read the contents of the file when the file is available, i.e. at runtime. Implementing such hook in the compiler was challenging but possible – the C++ language and the compiler had the necessary bits and pieces to do it.

The produced AST is still ill-formed and no code generation is possible. The second action that happens at compile time is to replace the ill-formed AST nodes with healthy ones and preserve the expected semantics (**Figure 5**).

Let's study the most trivial example – a standalone function call with no arguments to pass in. As explained above, the file "injects" its content in the current scope at runtime. Since there is not enough information at compile time the only way to fix the ill-formed expression is to delay its evaluation until runtime. That fix-up takes place in the AST by using an AST-to-AST transformation. It transforms the unknown expression into a specially crafted templated callsite. The callsite is a member call to the runtime embedded incremental compiler, i.e. to Cling itself. In other words the embedded incremental compiler is asked to compile the expression at runtime.

Handling the general case needs more information about the surrounding environment, i.e. the context it was seen in. The dynamic expression might be a subexpression or part of a statement. Let's consider **Figure 6**:



```
TFile* F = TFile::Open("demo.root");
int a,b = 1;
if (hist->Add(a, b)) {...}
...
```

```
TFile* F = TFile::Open("demo.root");
int a,b = 1;
if (gCling->EvaluateT("h->Add(*(int*)@, *(int*)@)", (void *[2]){ &a, &b }, ...)) {...}
...
```

**Figure 6.** A complex example of dynamic expression.

Usually the dynamic expressions are used homogeneously in the source code. We must allow mixing static and dynamic expressions. A main issue when mixing is the storage sharing. (For example, use static variables in dynamic expression). It leads to some extra implementation challenges at compilation time. Proper handling of mixed static and dynamic expression needs a way of sharing the relevant context. Cling does it by embedding the necessary information in the string that is passed for recompilation.

In the particular type information and storage information of the actual arguments have to be passed in. The storage information is presented as void pointers showing the memory address of the arguments. The type information is needed for casting the void pointers to the expected arguments' type. Just before recompilation happens the "@" placeholders are replaced with the actual arguments' addresses in memory.

The construct that contains the dynamic expression is important as well. When the expected return type of the callsite is recognized, the callsite template instantiation happens programmatically. In the snippet in **Figure 6**, the condition of the "if"-statement expects value of type Boolean. Thus the callsite is instantiated as Boolean. Later on, the recompilation makes sure the real result matches the expectation of the dynamic expression and if not, a diagnosis is issued.

### *4.3. From error recovery to code unloading*

A very important feature, which improves the interactivity, is code unloading. A common scripting technique is to load a script, test its functionality and if there is any logical mistake, unload it, fix it and load it again (**Figure 7**). It is unthinkable for a compiler library and therefore a challenging problem to tackle.

The semantics of the code snippet in Figure 7 suggests that a part of already executed code could be removed and even replaced with a different one. Introducing it in Cling is a big challenge, because it requires many transformations of the code and its representations. It happens at two different levels: the underlying compiler (Clang) and the low-level LLVM bitcode. The implementation of code unloading relies on the error recovery subsystem. It adds special restore points, which remember the transaction it has to restore to. The error recovery subsystem is used to clean up compiler high-level structures (like the AST).

```
[cling]$ .L Calculator.h
[cling]$ Calculator calc;
[cling]$ calc.Add(3, 1)
(int) 2
[cling]$ .U Calculator.h
[cling]$ .L Calculator.h
[cling]$ Calculator calc;
[cling]$ calc.Add(3, 1)
(int) 4
```

**Figure 7.** Error recovery example.

Then the corresponding code generated for the transactions has to be removed from the code module. After the module is sanitized then it is recompiled. Note the restore points are inserted in such way so the dependency analysis is simplified.

*4.4. Extras*
A great advantage of reusing a world-class compiler library, such as Clang, to build an interactive interpreter is that we can take advantage of all the features it offers. Because of its design, Cling benefits from world-class optimizations, production-grade parser and compiler-grade correctness. We can use Cling as interactive interpreter not only in C++ and C but also with little effort OpenCL, ObjectiveC and ObjectiveC++ can be added. Cling can benefit and embed many of the tools using LLVM, such as AddressSanitizer[13] and Clang Static Analyzer[14] tools.

One can see that using Clang and LLVM as libraries help us to build complete interactive C++ interpreter with far less effort than building it from scratch. In total Cling's codebase is 10 000 lines of source code. This allows us to maintain only that part of the code and delegate the rest of the maintenance to the LLVM community.

## 5. Integration in ROOT

Because of the fact that C++ doesn't offer thorough reflection and type information for the objects, ROOT enhances it by fetching their description from the embedded interpreter. A description of an object usually denotes relations such as: what class the object belongs to; what are its members; what is its memory layout; etc. Having access to this information enables the framework to implement automatic IO subsystem, which is used extensively for serialization and deserialization of objects. Reflection is the ability to examine and modify the behaviour of an object at runtime. Many core features from processing signal/slots to pyROOT[15] are implemented using the technique (**Figure 8**).

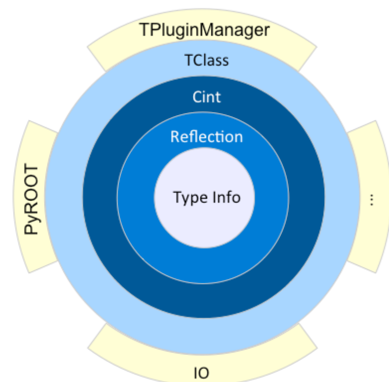



**Figure 8.** ROOT core.

The description of objects is kept in automatically generated files called dictionaries. Their total size for ROOT itself varies depending on which libraries are enabled and the target platform. A standard Linux built creates around 200 dictionary files consisting of approximately 1 million lines of C++ code. A single dictionary contains several distinct types of information, logically separated in four sections (**Figure 9**):

- TClass section – responsible for filling in ROOT's abstract interface, providing reflection abilities.

- ClassDef/Shadow section – extending TClass with several handy functions (such as ShowMembers), defined in a macro. If no ClassDef section is presented, a Shadow section is emitted in the dictionary. Shadow section helps calculating the memory layout of the objects.
- Call stub section – responsible for calling functions into compiled code. The call stubs are wrappers taking fixed set of arguments with different names and bodies. The call stubs are generated for each function in the described object. This causes a huge variant explosion if the function is template function, because all possible variants for a template instantiation have to be generated.
- Reflection information section – responsible for feeding proper information about the dictionary contents to Cint. Embodies a type injector, registering the dictionary contents in Cint's type database.

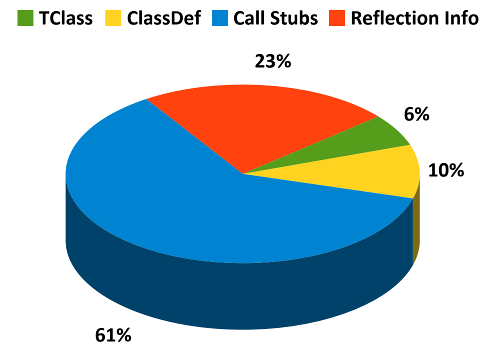



**Figure 9.** Dictionary section proportions.

As already mentioned, the embedded C++ interpreter shares the necessary information with ROOT. Sometimes the information has to be taken from interpreter's deepest internals, which introduces unwelcome implementation dependence. The framework abstracts out the implementation details of Cint by adding a new layer above – TClass. Nonetheless, there are places where the layer remains unused. Some of them are due to performance consideration, some are due to absence of the necessary interface in TClass and some just need replacement.

ROOT integrates Cling as its default interpreter in its new major version – ROOT 6. Cling comes with an excellent internal representation of the type information. The new C++ interpreter will leverage the reflection and type information in ROOT, solving many issues introduced by Cint. Consequently, there is no need of such form of compiled dictionaries anymore.

The scheduled set of changes is rather vast and requires considerable effort to make a smooth transition. The plan is to gradually switch between Cint and Cling by reimplementing the interfaces step-by-step. The concrete change plan for the dictionaries is to gradually reduce their size by removing sections until they completely disappear. The JIT compiler will replace call stub section naturally. Reflection information is not needed anymore by definition. Forthcoming proper target information allows ClassDef and Shadow section to go away, too. The TClass specific code will be generated on demand at runtime.

Migration towards Cling is a huge relief from the maintenance point of view in comparison to Cint. Cling's codebase is about 10 000 lines of source code as opposed to 230 000 for Cint.

Considering Cling's true nature it is very similar to ACLiC[16], i.e. it compiles in memory and it is binary compatible with GCC. This opens up entirely new world, waiting to be explored. Embedding the new interactive interpreter will help many parts from ROOT to be improved. Vivid examples are:

- Expression evaluation in TFormula and fitting;
- Interpreted classes deriving from compiled classes;
- Runtime optimizations in IO framework.

## 6. Conclusion and future work

The advancement of the software technologies allows us to build up huge and complex binaries – such as libraries, frameworks and tool chains. A modern application uses many external binaries, which makes its development sometimes a nightmare. Very often developers have to deal with tens of external binaries and one of the most difficult tasks is to find the right entity (eg. constant, variable, function), which offers the expected result. Occasionally the lack of documentation forces the

developer, in order to understand what a function does, to look at its implementation. This might be an obstacle for closed source externals. The slow but steady evolution of the programming languages adds one more dimension of complexity.

Rapid prototyping of software requires a systematic software development methodology and user-friendly tools. Flexible, easy to use and easy to learn, interactive environments are highly suitable for such kind of software prototyping.

Cling, Clang and LLVM are a huge leap in improving developers' comfort. Besides their modularity and reusability they could be stable ground for further improvements of existing development tools.

Developers' comfort and experience can always be improved. There are two general directions: proving handy tools and extending the programming language.

For instance, Cling terminal could bound to an interactive development environment (IDE) and execute the code on the fly in a sandbox. An IDE is the perfect environment where more tools could be integrated to help rapid application development. Work could be done in enabling auto-completion, static analysis and memory checkers.

Many of the programs, especially in HEP, contain great amount of iterations over containers. It becomes very tedious to extract the relevant data out of the containers. In its essence many of the iterations can be represented with unions and intersections of sets, i.e. algebra of sets. Embedding such declarative (SQL-like) type of language in Cling would greatly improve the rapid prototyping. LINQ[17] is a perfect example and proof.

## 7. References

[1] Martin J 1991 *Rapid Application Development* (New York, NY: Macmillan)
[2] Lattner C and Adve V, M 2004 LLVM A Compilation, Framework for Lifelong Program Analysis & Transformation *In Proc. of CGO'04*
[3] Brun R and Rademakers F, S 1996 ROOT - An Object Oriented Data Analysis Framework *In Proc AIHENP'96 Workshop*, Lausanne, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) pp 81-86
[4] Goto M, A 1996 Concept and application of Cint C++ interpreter *Interface magazine* (Japanese)
[5] Aho A, Lam M, Sethi R and Ullman J 2007 *Compilers: Principles, Techniques, and Tools (2nd Edition)* (Addison-Wesley)
[6] Sandewall E M 1978 Programming in an interactive environment: the "Lisp" experience. *In Computing Surveys* **10(1)**, pages 35-7
[7] CSharpRepl, http://www.mono-project.com/CsharpRepl (visited June 2012)
[8] Cheng H 2003 The Ch Language Environment *User's Guide. Revision 3.7.* (SoftIntegration, Inc. Davis)
[9] Standard ECMA-335 Common Language Infrastructure (CLI), http://www.ecma-international.org/publications/standards/Ecma-335.htm
[10] Clang: a C language family frontend for LLVM, http: //clang.llvm.org/ (visited June 2012)
[11] LLDB: a LLVM debugger, http: //lldb.llvm.org/ (visited June 2012)
[12] C++ Standard ISO/IEC 14882:2011(E) 2011
[13] Address Sanitizer: a fast memory error detector, http://code.google.com/p/address-sanitizer/ (visited June 2012)
[14] Clang Static Analyzer: a source code analysis tool, http://clang-analyzer.llvm.org/
[15] Generowicz J et al., Reflection-based Python–C++ bindings, *In LBNL Papers* http://repositories.cdlib.org/lbnl/LBNL-56538/
[16] Antcheva I et al., 2009 ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization *In Proc of CPC* **180**(**12**), pages 2499–2512
[17] LINQ: Language Integrated Query, http://msdn.microsoft.com/en-us/library/bb397926.aspx